ChatGPT

Comprehensive Plan for AWS Migration and Multi-Tenancy Enhancements

1. Containerization and AWS Infrastructure Setup

Containerize the Replit application: Begin by creating a Dockerfile for the Flask app. This image should include all dependencies (Python 3.11, Flask, Gunicorn, Tesseract OCR, etc.) as specified in the Replit environment. Test the container locally to ensure the app runs via Gunicorn on the expected port. This container will provide a consistent runtime for deployment.

AWS deployment environment: Choose an AWS service for running the container. A recommended approach is to use **AWS ECS Fargate** for a serverless container deployment, or alternatively **AWS Elastic Beanstalk** (Docker platform) for simplicity. For fine-grained control and easier scaling, ECS with Fargate is ideal. Set up an AWS ECR (Elastic Container Registry) and push the Docker image to ECR. Then create an ECS Task Definition referencing that image and exposing port 5000. Configure an ECS Cluster and Service (with Fargate) to run the task. Attach an **Application Load Balancer (ALB)** to the service for HTTPS traffic routing. The ALB will listen on port 443 and forward requests to the container's port 5000.

Networking and security: Deploy the ECS service in a new VPC with private subnets for containers and a public subnet for the ALB. Use security groups to allow inbound HTTPS to the ALB and restrict container access (e.g. only ALB can talk to containers, containers can talk to DB/Redis). Enable AWS Certificate Manager (ACM) to provision an SSL certificate for your domain and attach it to the ALB so that all traffic is encrypted in transit. This satisfies the TLS 1.2+ requirement for HIPAA compliance [1].

Environment configuration on AWS: On ECS/Beanstalk, configure environment variables and secrets required by the app. This includes Flask settings, any API keys, and especially the Epic OAuth credentials (client IDs, etc.). Use AWS Secrets Manager or SSM Parameter Store for sensitive values like `EPIC_CLIENT_SECRET` and database passwords, loading them into the container's environment at runtime. Ensure the Flask app reads these from the environment (this may already be how Replit was configured via Replit Secrets). Also include an environment variable for the proper **FLASK_ENV/CONFIG** to switch to production settings (e.g., turning off debug mode, using the production database URL, etc.).

Database and Redis hosting: (Details in the next section.) The key point for infrastructure is to **not host the database or Redis inside the container**. Instead, use managed services: AWS RDS for Postgres and AWS ElastiCache for Redis. The container will connect to those via environment variables (hostnames, credentials). This separation allows independent scaling and persistence.

CI/CD pipeline: Set up a deployment workflow to streamline updates. For example, use GitHub Actions to build the Docker image on each push, run tests, then push to ECR and deploy the new task definition on ECS. This ensures that future changes to the app can be rolled out on AWS quickly and reliably (optionally with zero-downtime if using ECS rolling updates).

Domain name and DNS cutover: To avoid needing a new Epic app registration, use (or obtain) a custom domain for the application rather than a Replit-specific URL. For instance, if you have `app.healthprep.com` registered in Epic as the redirect URI, continue using that. Point this domain to the AWS ALB (using Route 53 DNS or your DNS provider) and verify the SSL certificate. During migration, you can add the new AWS endpoint as an **allowed redirect URI** in the Epic app configuration (many OAuth configs allow multiple redirect URIs) to facilitate testing. Ultimately, update DNS so that the custom domain now resolves to AWS. Because the domain stays the same, Epic will continue to recognize the redirect and **no new registration should be required** – the client ID and secret remain valid, just the hosting environment changed. If a custom domain was not in use, consider **registering one and updating Epic** ahead of time to that domain to enable a seamless switch. Otherwise, coordinate with Epic's App Orchard or the client's Epic team to update the app's redirect URL to the new AWS domain to avoid a full re-registration.

Logging and monitoring: Enable CloudWatch Logs for the ECS containers to aggregate application logs. Configure CloudWatch alarms for critical metrics (CPU/memory usage of containers, response latency, etc.) to ensure the 99.9% uptime goal is met. Optionally use AWS X-Ray for tracing requests, which can help pinpoint performance issues in the app. Set up AWS CloudTrail and other security monitoring (GuardDuty) for compliance – this aligns with breach detection and audit requirements (e.g., monitoring for unauthorized access) [2].

Scaling and redundancy: The AWS setup should support scaling out if usage grows. For ECS, configure auto-scaling policies (e.g. add another task/container if CPU > 70% for 5 minutes) so multiple containers can run behind the ALB. With the database on RDS (multi-AZ) and stateless app containers, you can achieve high availability. Also consider running at least 2 ECS tasks in different AZs for redundancy. Ensure sessions are handled properly in a multi-instance scenario (e.g., use a shared session store or sticky sessions; more on this in section 3 under session management).

2. Database Migration and Architecture Improvements

Migrate from Replit PostgreSQL to AWS RDS: Since the application already uses PostgreSQL as its primary database [3], spin up an **Amazon RDS for PostgreSQL** instance. Choose an instance size that fits current needs (e.g., db.t3.medium for development, scaling up as needed in production) and enable Multi-AZ deployment for failover. Enable automated backups and set a retention period (e.g., 7-14 days) to allow point-in-time recovery. Make sure to enable encryption at rest for the RDS instance (AWS uses AES-256 for RDS storage encryption [1]). Export the data from the Replit Postgres (e.g., using `pg_dump`) and import it into the RDS instance before cutover. You can run the migration scripts (like `migrate_multi_tenancy.py` if not already applied) on the new database to ensure the schema is up to date [4]. Update the application's database connection string to point to the new RDS endpoint, and test connectivity from the container (likely done via environment variable e.g. `DATABASE_URL`).

Connection management: On AWS, network restrict the RDS instance to only accept connections from the app's security group or VPC – no public access. This keeps the database hidden from the internet. Use a connection pool in the application (if not already configured via SQLAlchemy) to optimize database use across multiple app threads/workers. If needed, AWS RDS Proxy can be introduced later to manage a large number of connections efficiently.

Data isolation and multi-tenancy improvements: The schema already supports multi-tenancy by scoping data with an `org_id` field on all key tables (patients, users, documents, etc.) [5] [6]. To further improve this: - **Indexes & Performance:** Ensure all frequently queried tables have composite indexes including `org_id` (the models have added indexes for org_id and other fields as seen in Document and FHIRDocument models [7] [8]). This will keep queries fast as data grows and multiple organizations' data intermingle. If some tables will grow very large (e.g., documents, audit logs), consider partitioning them by `org_id` or date to improve query performance and maintenance. PostgreSQL could partition by organization ID, so each tenant's data is in a separate partition – this can speed up queries by pruning partitions, though it adds complexity in management. - **Row-Level Security (optional):** For an extra layer of protection, you could enable PostgreSQL Row Level Security policies to enforce `org_id = current_org` on queries at the database level as well. However, since the application logic already enforces isolation, this is a defense-in-depth improvement to consider later.

Scalability considerations: As the number of tenant organizations grows, the single database may become a bottleneck if one tenant generates disproportionate load. The current plan is to use one RDS instance for all tenants, which is simplest for now. In the future, be prepared to **scale vertically** (choose a larger DB instance) or **horizontally** partition data: - A potential strategy is **schema-based multi-tenancy**: Each organization could have its own schema within the same database (requiring code changes to set the schema per session). This offers clearer separation but complicates queries and migrations. - Another strategy is **database per tenant** (each large client on a separate RDS instance). This is easiest for data isolation but would require the app to handle multiple connections and doesn't scale well to many small tenants. For now, with the moderate number of tenants, the single database with org_id filters is acceptable [9], and you can revisit these strategies as needed when scaling to many clients or very large data volumes.

Database optimization and maintenance: Leverage RDS features like Performance Insights to monitor query performance per organization. This can identify if one org's usage patterns (e.g., a heavy report or sync) are impacting others, and then you can optimize those queries or add caching. Set up a regular maintenance window for the RDS (updates, minor version patches) and ensure you have a **snapshot backup policy** beyond automated backups (e.g., monthly snapshot stored long-term) for compliance. Also prepare a **data retention policy**: since HIPAA requires 6-7 years retention for audit logs, you might eventually archive older data (perhaps moving old audit records to S3 or Glacier). But in the near term, the Postgres DB will hold all active data and the retention can be managed by simply keeping it until manual deletion.

Redis migration: The app uses Redis + RQ for background tasks [10]. Instead of relying on a local Redis (which may have been running on Replit), use **AWS ElastiCache for Redis**. Launch a small Redis node (in the same VPC) and update the RQ configuration (`redis_url`) to point to it. This provides persistent, highly available task queue storage. The Redis instance should also be in a private subnet, accessible only to the app. With this in place, background jobs (patient sync, prep generation, etc.) will survive app restarts and can scale. In production, you might run a dedicated **worker process** or container to execute RQ jobs (so the web app containers are not doing heavy background processing). For example, have an ECS Service for the web app and another ECS Service (or separate process) for running `rq worker` that listens to the queues. This way, you can scale background workers independently if needed for large batch jobs.

Encryption of sensitive data: Take advantage of AWS Key Management Service (KMS) or similar to manage encryption keys. In the database, fields like `EpicCredentials.access_token` and `refresh_token`

should be stored encrypted or tokenized. The code already intends to store encrypted tokens (per comments) – implement this by encrypting those values before saving to DB (for example, using Fernet symmetric encryption with a key stored in AWS Secrets Manager or using PostgreSQL's pgcrypto). This ensures that even if the database were compromised, PHI and credentials remain protected [1]. Similarly, if any PHI is stored in logs or files, ensure those are encrypted at rest (S3 can be configured to encrypt all objects, RDS and ElastiCache have encryption enabled). The RDS and ElastiCache services themselves, when configured, will handle encryption at rest, but application-level encryption of critical fields is an extra safeguard.

3. Improving Multi-Tenancy Structure & Organization Management

Tenant model and isolation: The application's multi-tenancy foundation is already in place via the `Organization` model and `org_id` scoping [9]. We will reinforce this structure and make it more robust: - Audit all data access in the code to ensure every query is properly filtered by `org_id`. Implement utility functions or SQLAlchemy query filters (as hinted by the `OrganizationScope` utilities [11]) to automatically apply the current tenant's ID to queries. This reduces the risk of developer error exposing cross-org data. - Add organization context to all in-app operations. For instance, when an admin from Org A is logged in, any creation of a patient or document should automatically stamp their `org_id`. The Flask routes can use a decorator (e.g. `@require_organization_access`) that checks `current_user.org_id` matches the resource's org_id, preventing tampering via URL changes or IDs not belonging to that org [12].

Root admin oversight: The system should have a **Root Administrator** (a user with `is_root_admin=True` or role "root_admin") who can manage all organizations. Build a root admin dashboard (likely `/root_admin` area) to list organizations, view their status, and manage them. This interface allows the SaaS owner to onboard new clients and support existing ones: - **Organization creation:** A root admin can create a new `Organization` record via a form (enter clinic name, contact info, etc.) [13]. Upon creation, if immediate payment or contract is not handled in-app, you might set the org's `setup_status` to "trial" or "incomplete" until they finish setup and payment. - **Admin user invitation:** Once an org is created, generate an **admin invite link** for that organization's first user [14]. This could be a secure token encoded link (e.g., containing org ID and an expiring signature). Email this link to the organization's contact. When they visit, allow them to set up their account (choose their password, possibly set up 2FA as well) and mark that user as `role=admin` for that org. This process ensures the first admin account is created securely without sending raw passwords. The invite link should be one-time use and expire after, say, 7 days if not used. - **Epic credentials setup:** After logging in, the new org admin should be guided to an Epic configuration page (the `/admin/epic/registration` or `/admin/epic/config` route). At this point, their org's record is empty of client ID/secret, so provide them with instructions on how to obtain these. For example, include help text or links: "*Register an OAuth client in your Epic environment or App Orchard. Provide the following redirect URL:* `https://app.yourdomain.com/fhir/callback` *(for example). Once Epic provides a Client ID and Client Secret, enter them below.*" Ensure that on this page we also link to Epic's documentation or your own knowledge base article for guidance. We need the **Client ID, Client Secret, FHIR base URL** from the organization. We may also ask for the **OAuth2 Authorization and Token endpoints** if their Epic installation uses custom endpoints (the code supports storing these in `epic_oauth_url` and `epic_token_url` per org). In return, we supply any information they need to register the app: - Redirect URI (our app's OAuth callback URL). - JWKS key or certificate if using any JWT authentication method (likely not needed for standard OAuth code flow). - The list of OAuth scopes our application will request (so they can enable those scopes for the app in Epic). - Our application's name and

description (they might need to provide it during registration). By providing these resources in the admin interface, each organization can self-service their Epic integration setup with minimal back-and-forth. **Include a FAQ or help modal on `/admin/epic/registration` for common steps to find Epic credentials.** (For example: *"Log in to Epic's App Orchard or contact your Epic support team to register a new OAuth client. Provide the redirect URL above. Once you receive a client ID and secret, enter them here. For Epic sandbox, use the default FHIR base URL provided; for production, get the specific FHIR base URL for your Epic instance."*)

- **Organization settings and limits:** Use fields like `max_users` (already in the model) to enforce user limits per plan. If an organization is trial or on a basic plan, we can cap `max_users` and check `Organization.can_add_users()` before allowing new user creation [15]. This ties into billing (e.g., higher plans can have more users). Also, use `setup_status` to control access – e.g., if an org is "incomplete" or "suspended (non-payment)", the admins/users might only see a message or have read-only access until resolved.

Admin and user management within a tenant: Once an organization's first admin is set up, that admin can manage their staff: - Provide an **Organization Admin Panel** (at `/admin/dashboard` for that org) where the admin can create and invite users under their organization. This is partly implemented through user role checks (only admins can create users) [16]. We will refine it so that an admin fills out a new user's name, email, and role (nurse, MA, or admin). Upon submission, the system sends an invite email with a link for that user to set their password (similar to the admin invite flow). This ensures passwords are not emailed and users choose their credentials securely. - Implement role-based access control (RBAC) consistently: the UI should hide or disable admin-only functions for non-admins, and the backend should enforce it. For example, only admins should see the "User Management" section, and routes like `/admin/create_user` should be protected by something like `@require_role('admin')` [17]. The roles defined (`root_admin`, `admin`, `nurse`, `MA`) each have specific permissions – ensure these are honored in every feature (the code's convenience methods like `user.can_manage_users()` [18] and `can_manage_organizations()` can be used). - **Two-factor authentication (2FA/MFA):** Given the sensitivity of PHI and security goals, enable two-step verification for user logins, at least for admins. The `User` model has a `two_factor_enabled` flag [19], so we will build out the feature around it: - Decide on a 2FA method: **Authenticator app (TOTP)** or **Duo Push** are two options. Using TOTP (e.g., Google Authenticator) is simpler: when a user enables 2FA, the app generates a secret and QR code for the user to scan, and subsequently, at login the user must provide the 6-digit code. This can be implemented with a library like PyOTP. The secret key should be stored encrypted for each user. - If using **Duo Push** (which provides a push notification to a mobile app), it requires integrating with Duo's API and iframe – an admin would enroll in Duo, and our app would trigger a push at login. This is more complex and requires a Duo account and application setup, but it provides a user-friendly experience. This could be offered for enterprise clients who have Duo. It might be done outside Epic (our app would directly work with Duo's service). - In either case, make 2FA optional per user but **mandatory for root admins** and perhaps highly recommended for all admins. You could enforce two-factor for any account with admin privileges by policy. During login, if 2FA is enabled, after the password check, present the second factor challenge (OTP code input or send Duo push and wait for success). - This addresses the security goal of MFA [20]. It is independent of Epic Hyperspace's 2FA – note that Epic's own user login 2FA does not protect our app, since our app is a separate system. So we implement 2FA on our side for our application's authentication. - **Additional identity verification questions:** As an extra security measure, we can add configurable security questions for user account recovery or verification. For example, when creating an account, a user could set answers to a couple of security questions. These could be used in a "Forgot password" flow to validate the

user before allowing a password reset, especially if email alone is not deemed secure enough. This is optional, but for admin accounts it could add another layer if needed. Initially, we might skip this unless there's a compliance requirement, since it adds user friction. But it's noted as a possible improvement for verifying identity if someone contacts support, etc.

Session and access management: Enforce secure session handling in the new environment: - Continue using Flask-Login for session management, but ensure SESSION_COOKIE_SECURE=True and SESSION_COOKIE_SAMESITE=strict in production config so that cookies are only sent over HTTPS and not accessible to other domains. This prevents attacks like session hijacking. - Because we will run multiple app instances, consider using a **server-side session store** (instead of Flask's default cookie-based session or local memory). For example, use Redis (we have ElastiCache) to store session data, or use database-backed sessions. This way, any instance can validate a session. Alternatively, configure the ALB to enable **sticky sessions** (session affinity) so that a user's requests go to the same container each time. However, a shared session store is more robust if we autoscale or replace containers frequently. - Maintain the session timeout (30 minutes by default) [19] [21] to log out inactive users, and implement absolute session expiration if required (e.g., force re-login after 8 hours even if active). Provide an admin setting for session timeout if different orgs have different security policies. - **Account locking and monitoring:** The system locks accounts after 5 failed attempts for 30 minutes [22]. In AWS, ensure emails or alerts could be sent if a brute-force is detected (e.g., many failed attempts). This ties into breach detection – such events should be logged (perhaps already via the audit log) and monitored by administrators.

Audit logging and compliance: The app has an audit logging system in place to track user actions and data access (HIPAA compliance) [23]. Strengthen this by: - Logging critical security events: logins, 2FA setup, password changes, privilege changes, failed login attempts (with IP), etc. These logs should include org_id, user_id, timestamp, and details, and be immutable. Consider using a separate database table or even an external log system for audit logs to ensure they aren't tampered with. The `enhanced_audit_logging.py` and `log_admin_event` functions likely cover some of this – verify they record everything needed. - Consider streaming audit logs to a secure location (for example, to CloudWatch Logs or an S3 archive) for long-term storage [24]. This provides a backup beyond the primary database, and you could even use AWS Athena to query those logs if needed in an investigation. - Implement an **admin audit viewer**: allow organization admins to view logs pertaining to their org (e.g., which user accessed which patient record at what time) to fulfill accounting of disclosures. The root admin should be able to view all orgs' logs for oversight. - Ensure that log retention meets the required duration (7 years is mentioned for HIPAA). AWS can lifecycle old log files to Glacier to satisfy long retention cost-effectively.

Data partitioning per client (future consideration): If a large hospital client demands absolute separation, we could deploy a dedicated instance of the app for them (their own container and database). This is outside the primary multi-tenant approach but can be done if needed. Mention this as an option for enterprise clients: our architecture is flexible to either **host multiple tenants in one environment** or **spin up isolated stacks** if required by a client's IT policy. Using infrastructure-as-code (like Terraform or CloudFormation) can help duplicate the environment for such isolated deployments.

4. Secure OAuth Integration with Epic (Seamless Epic Configuration)

Information exchange for OAuth: For each tenant organization to use our software with their Epic system, both parties need to share certain information securely: - **What we need from the organization (Epic client):** The admin will input their Epic **Client ID** and **Client Secret** into our system (these come from the

Epic App registration for our app). They will also provide the **FHIR Base URL** for their Epic environment (for Epic sandbox it might be the standard shared endpoint, for production it will be the unique URL for that health system's FHIR API). Additionally, we capture the **Epic OAuth2 authorization URL** and **token URL** if their environment provides unique endpoints (often Epic's OAuth endpoints can be derived from the base URL, but having them explicitly is helpful). Fields like `epic_endpoint_id` and `epic_organization_id` in our model can record identifiers from Epic's end (e.g., an App Orchard ID or the organization's Epic identifier) for reference if needed [25]. - **What we provide to the organization:** We must supply the **Redirect URI** that they should configure in Epic. By default, our app likely uses something like `https://<our_domain>/fhir/callback` or `/oauth/callback` for the OAuth code exchange – confirm the exact path (perhaps defined in the routes as the Epic OAuth authorize callback). The admin should use this in their Epic app configuration. If we use a custom domain, this URI will remain constant across environment moves, which is why using the custom domain is crucial for not re-registering. We also provide the **scope** we need (e.g., `patient/Patient.read`, `patient/DocumentReference.read`, etc., and offline_access for refresh tokens). Ensure the admin knows to enable these scopes for the client in Epic. If our app uses a **JWKS (JSON Web Key Set)** for token verification (some SMART on FHIR apps register a JWKS URL for verifying JWTs), provide the JWKS endpoint URL as well. Typically, though, for OAuth clients Epic issues tokens to us; we might not need a JWKS unless we are acting as an issuer – likely not the case here. - **App registration details:** If the organization is using Epic's App Orchard (for a public app) or their internal App Registry, they might need the **App Name and description** to locate it or for auditing. We have fields for `epic_app_name` and `epic_app_description` to store what was submitted [26]. We should populate those for record-keeping, and perhaps provide a reference number if Epic assigns one upon approval.

Establishing the connection: Once the org admin enters their Epic credentials and hits save, our app will store them (client secret encrypted) and mark the org's Epic environment (sandbox or production). The admin can then click "Connect to Epic" which triggers the OAuth flow. In a user-context OAuth (SMART on FHIR), it would redirect them to Epic's authorization page to log in and authorize. However, if this app is more service-account based, they might use client credentials flow. Given the interface, it likely does a user authorization (since it mentions a login via MyChart). So: - For **sandbox testing**, we use the shared Epic sandbox credentials (maybe a global client ID) or the org's provided sandbox ID if they have one. - For **production**, once the client ID/secret are in, when the admin clicks Connect, it redirects to their Epic login (at `epic_oauth_url`) and after they authenticate, Epic redirects back to our app with an auth code which we exchange for tokens. We then save the tokens in `EpicCredentials` (tied to the org, possibly also tied to the user who did it). - After connection, mark `Organization.is_epic_connected=True` and store the token expiration etc., so that the admin (and our system) knows the integration is live.

Maintaining the connection without re-registration: To avoid re-registering the app after moving to AWS, ensure that **the Client ID and secret remain the same**. These are issued by Epic when the app is initially registered. Since those are stored per org in our DB, migrating the database will carry them over. The crucial part is the redirect URL: Epic will only redirect to the URI that was whitelisted. If our domain changes and is not whitelisted, the OAuth will fail. Therefore: - We have already planned to use the same domain name on AWS. By doing so, from Epic's perspective nothing changed – the redirect to `https://app.healthprep.com/fhir/callback` will still work, it just goes to the new host. - If any aspect of the app's **launch URI** or **JWKS URI** changed (for example, if on Replit it was a `.repl.co` domain and on AWS it's custom), we must update those in Epic's records. Ideally, do this update ahead of time. Epic's App Orchard allows editing an app's settings (for sandbox apps it's self-service; for production one might have to go through their support). - Because the user specifically mentioned not having to "re-register" with Epic, our plan ensures **continuity of these endpoints**. We will coordinate a cutover such that the Epic

configuration is updated (or already set) for the new endpoints before shutting down the old one. If Epic allows multiple redirect URIs, add the new one, test it, then remove the old one once migration is done.

Security of OAuth credentials: We treat the Epic client secret like any other secret – do not log it, store it encrypted, and restrict access. On AWS, you might choose to store the client secrets in AWS Secrets Manager rather than in our database, but since our app already has a model for it, we will continue to use the database (encrypted at rest). We can improve by **encrypting at the application level** as noted (using KMS or a symmetric key). Also, use short token lifetimes if possible and rely on refresh tokens. The `epic_token_scheduler.py` likely handles refreshing tokens periodically so that the system can operate without constant user re-authentication. Ensure this scheduler (or background task) is running on AWS (could be a cron job, or an RQ scheduled job) to refresh tokens before they expire. This prevents disruption in service (and avoids requiring user to re-initiate the OAuth frequently).

Epic Hyperspace vs our app 2FA: The question about two-step verification via Epic Hyperspace suggests whether the admin's login to Epic could serve as 2FA. In our design, when an admin connects to Epic (through the OAuth flow), if Epic requires multi-factor for that user, Epic will handle it during the OAuth login. But that is separate from our app's login. Our admin still should secure their login to our app with MFA as discussed. So effectively, an admin might go through two MFA processes: one for our app (if enabled) and one when logging into Epic's portal during OAuth. This is acceptable because they are distinct contexts. We will clarify this in documentation to avoid confusion: *"Enabling 2FA in HealthPrep ensures secure access to the admin dashboard. This is independent of any Epic Hyperspace login requirements. During the Epic data authorization step, users may also need to authenticate with Epic (which could include Epic's own MFA if configured)."*

Testing and validating OAuth on AWS: Before going live, use the Epic sandbox to test the full SMART on FHIR flow on the AWS deployment. Ensure that the redirect URIs, CORS settings (if any), and network connectivity (the app should be able to reach Epic's FHIR server endpoints) are all working. Debug any issues in a staging environment. This will catch things like missing HTTPS (Epic will require redirect URIs to be HTTPS), or firewall issues preventing our app from calling Epic's APIs. Only once the sandbox tests succeed do we proceed to production use.

5. Admin and User Onboarding Workflow Enhancements

Automated organization onboarding: Streamline the process of getting a new clinic or organization onto the platform: - As mentioned, use a root admin tool to create an org and send the invite. To implement the **invite link generation**, create a secure token (could be a JWT or a signed serializer token) that encodes the org_id and perhaps an expiry time. For instance, generate a random token stored in a small "Invite" table or include it in the URL and store a hash in the org's record. When the invite URL is hit, the app verifies it, asks the user to set up their admin account (name, email, password), and perhaps some security info (2FA setup or security questions if required). After successful setup, mark the invite as used so it cannot be reused or replayed. - Use **AWS SES (Simple Email Service)** or a trusted email API to send out the invite email to the organization's contact. The email should be professional and include the one-time link. We should also prepare templates for other emails (user invites, password resets, etc.) as the app grows. - Once the admin registers, prompt them through initial setup steps. We can implement an **onboarding checklist UI** for new admins: e.g., "Step 1: Enter Epic credentials. Step 2: Configure your screening presets or import defaults. Step 3: Invite your staff users. Step 4: Start first sync." This makes sure they get value quickly. Some of these steps can be optional or skippable, but showing progress can improve the onboarding experience.

User creation and management: For adding subsequent users (nurses, MAs, secondary admins): - The admin panel should list current users of the org and their roles/status. Admin can deactivate users (set `is_active_user=False` if someone leaves) or promote a user to admin (if allowed – perhaps only root admin can designate another admin? Or organization admin can create other admins within their org). - The invite flow for regular users is similar: admin clicks "Invite User", enters name, email, role; the system sends an email with a one-time registration link where the user sets their password. This approach is more secure than the admin setting a password for them. We must handle if the invite expires or is not accepted – allow admin to resend or cancel invites. - If using 2FA in the system, consider **forcing new admins to set up 2FA on first login**. For other roles, make it optional or based on org policy. We could allow the organization admin to toggle "Require 2FA for all users" in their settings if they desire higher security across the board.

Two-step verification (Admin perspective): If we integrate Duo for 2FA, an admin might need to register their device with Duo. The onboarding could include "Enable Duo 2FA" step where the admin scans a QR code or gets a push to link their account. However, implementing Duo might be involved, so a simpler initial solution is TOTP as outlined. The admin interface can show "Two-Factor Authentication: Enabled/Disabled" and allow them to manage their 2FA (view QR code, generate backup codes, etc.).

Identity verification questions: We touched on adding security questions for password reset. If we implement a self-service password reset, one flow is: - User clicks "Forgot Password", enters email. - If email is found and user has security questions on file (and maybe if 2FA was enabled), we can prompt one of the questions. - If answered correctly, send a password reset email or allow them to reset directly. This reduces the chance of email alone being used to compromise an account, but it's optional complexity. We might decide instead on a simpler approach: password resets via email link (and if the account has 2FA, require the OTP after clicking the email link to finalize the reset).

User experience and UI: All these flows (invites, registration, 2FA setup, etc.) should be built into the web interface with clear guidance. Use modals or separate pages for these steps with Bootstrap styling consistent with the app. Also, ensure that error messages (expired link, invalid token, etc.) are handled gracefully with instructions to contact support or resend invite.

Admin "Hypothetical initiation" example: Summarizing the flow: 1. **Org Creation:** Root admin creates "Clinic ABC" in system, enters contact email of Dr. X (the person who will be the first admin). 2. **Invite Email:** Dr. X receives an email: "You've been invited to HealthPrep as the admin of Clinic ABC. Click here to set up your account." This link is one-time and directs to our app. 3. **Admin Registration:** Dr. X clicks and is taken to a secure registration form. They set their password (and possibly are prompted to set up 2FA right after password creation, if we enforce that). Their account is created with role=admin and linked to Clinic ABC. 4. **Post-registration:** The app might automatically log them in or direct to login. Upon first login, they see an onboarding page or checklist. They complete Epic OAuth setup: entering Client ID/secret and initiating the auth flow to connect the app to Epic. The system confirms "Epic Connected!" [27] [28] or shows any errors if credentials were wrong. 5. **Populate data:** Once connected, the admin can trigger an initial patient data sync (perhaps fetch a test patient or upcoming appointments). They can also navigate to presets or settings. 6. **Invite staff:** The admin goes to User Management, enters emails of staff (nurses/MAs) to invite. Those users get their emails and join similarly. 7. **Ready to use:** Users can now log in, see patients and prep sheets for their organization only. The admin can always return to the Epic config page to update credentials or see connection status [29] [30], and to the billing page if needed (next section).

By planning this flow out, we ensure that when we migrate to AWS and open for more organizations, the process is smooth and mostly self-service, which is crucial for scaling to many clients.

6. Billing and Subscription Model Integration

To turn the application into a SaaS offering, integrate a payment and subscription system (e.g., Stripe) so that organizations can sign up for paid plans. Key points for billing:

- **Plan definition:** Define pricing plans, for example:
- Single clinic plan: \$100/month for up to X users and Y patients (as an example).

- Group/hospital plan: tiered pricing (e.g., \$X per user or bulk discounts at certain user counts), or custom enterprise pricing. We want to incentivize bulk purchases, so perhaps offer packages like "10 clinics for \$Y/month" or discounted rates for larger groups. This can be handled via Stripe volume discounts or separate products.

- **Stripe integration:** Use Stripe's subscription APIs to create products and plans. For each organization, you might have a Stripe Customer entry. The workflow could be:

- During org onboarding, after the admin enters initial info, if they are to start a paid plan (as opposed to a free trial), direct them to a **Stripe Checkout** page for the chosen plan. This can be a link or embedded form where they enter payment details securely (the app itself should not handle raw credit card info).
- On successful payment, Stripe will create the subscription and can send a webhook to our app. We should handle the webhook (Stripe can call an endpoint on our app) to activate the organization. For example, once payment is confirmed, set `setup_status='live'` and maybe record the Stripe subscription ID in the org record for reference.
- If the user opts for a free trial first, mark `setup_status='trial'` and set `trial_expires` date 14 or 30 days out. We'll need to send them reminders to add payment before trial ends. This can be via email or a banner in the app.

- **Bulk pricing implementation:** If an organization represents a physician group that wants to onboard multiple clinics under one umbrella, we have a couple of approaches:

- **Single Org, multiple sites:** We could allow one Organization in our system to actually correspond to a group of clinics. They would use one Epic client ID if they share an Epic instance, or possibly multiple credentials if each clinic has separate Epic (this gets complex). Alternatively, we treat each clinic as a separate Organization in the app, but offer a discount for groups.
- The simpler way is to treat each Epic client (each installation) as one org in the app. For a hospital system with multiple sites but one Epic, that's one org in our app. For a company that owns 5 clinics each with their own Epic, that would be 5 orgs. We can then offer **bulk purchase** by linking those orgs at the billing level (e.g., a parent account).
- Implement a "parent customer" concept: the group buys 5 licenses on Stripe (maybe a custom plan), and we mark 5 orgs as paid under that customer. Or simply give them a coupon code to use for each clinic's subscription. This may not need code changes beyond tracking which orgs are part of a group and adjusting billing logic.

- In the interim, easiest might be to handle bulk deals manually (the root admin could mark certain orgs as paid or give discount codes). But as an advanced plan, design the data model to possibly have a linkage between organizations for group management or at least a field like `parent_org_id` or a separate table for group subscriptions.

- **Payment confirmation flow:** Ensure the admin portal has a section for **Billing** where an admin can see their subscription status (trial, active, past due, etc.). If using Stripe's customer portal, you can provide a link for them to update payment info or view invoices. If not, at least display next payment date or trial expiration. Also, enforce in code: if an org is `suspended` due to non-payment, prevent normal use (perhaps lock logins or show a "Please update payment to continue" message). The `setup_status` field in Organization can be used: e.g., set to 'suspended' if Stripe subscription is canceled or payment failed, and check that in login or main views to restrict access.

- **Stripe Webhooks:** Implement webhook endpoints to handle events like `invoice.payment_failed`, `invoice.paid`, `customer.subscription.deleted`, etc. For instance, on payment failure, notify the org admin via email and set a grace period. On subscription cancellation, mark the org as suspended. On successful payment (if coming out of trial or suspension), mark as live.

- **Bulk purchase incentive:** We may incorporate this in marketing and manual handling initially. For example, if a hospital network approaches us to onboard 20 clinics, we might not have an automated flow for that; instead, we'd handle it through sales and then set them up directly in the database or via admin tools. Still, the system should be flexible:

- Possibly allow the root admin to override `max_users` or apply a special pricing flag. For now, document these manual steps.

- Down the line, a more sophisticated approach is to allow an organization to add multiple Epic credentials for multiple sites under one account (that would be a major extension to the data model, essentially a hierarchy of organizations). This is beyond the immediate scope, but worth noting as a future roadmap item.

- **Preventing individual clinics from opting for small plan if part of group:** This is more of a business process – if selling to a big group, you'd likely manage their onboarding directly. But to enforce, you could ask during sign-up "Are you part of a larger network with a group plan?" and use some code or ID to tie them. However, that might not be necessary right now.

- **Pricing transparency:** Perhaps incorporate into the app usage stats like "X hours saved" or "Y screenings automated" for the admin to see the value (as per business goals [31]). This can help justify the subscription cost and encourage bulk upgrades. This is a nice-to-have analytic feature for the future.

In summary, integrating Stripe (or another payment gateway) will automate the monetization. We ensure that from the software side, an org's status (trial/active/suspended) is always checked to permit or restrict access, and that upgrading/downgrading flows are smooth. These changes prepare the app to handle real clients in a sustainable business model.

7. Ensuring a Seamless Epic Migration (Avoiding Re-Registration)

Before finalizing the move to AWS, take steps to ensure Epic integrations continue without interruption:

- **Domain consistency:** As emphasized, maintain the same external URLs that Epic knows about. If the Replit app was accessible at a replit.com domain and that was used in Epic, it's worth migrating to a custom domain now. For example, set up `https://healthprep.app` or similar while still on Replit (Replit allows adding custom domains). Update the Epic app's redirect URI to this custom domain *while the app is still running on Replit*. This way, you've decoupled the domain from the host. Then when switching to AWS, you simply repoint the DNS. Epic will still send tokens to `healthprep.app/callback` and our AWS-hosted app will handle it. No new client ID or approval needed.

- **Epic client ID scope:** Confirm whether each organization uses a **separate client ID** (probably yes, each org registers the app in their Epic instance) or if there is one global client ID for a multi-tenant scenario. It sounds like each has their own (since Org model stores epic_client_id per org). That means multiple client IDs exist across tenants. We don't want to break any of them. So for each one, ensure their redirect URIs are updated appropriately. Likely, all will use the same pattern of our domain, so just ensure that domain is accepted. Communicate with clients if needed to add the new domain as allowed redirect if their Epic has restrictions.

- **JWKS and SSL certs:** If Epic's OAuth flow involves validating our SSL or a JWKS, ensure the SSL certificate on AWS is from a trusted CA (ACM provides that). If we previously provided Epic with a public key for a backend service authentication, provide an updated one if that changed (though in our case, it's standard OAuth, so probably not applicable).

- **Testing token exchange:** After migration, do a test OAuth authorization for each type of environment (one sandbox, one production org) to be absolutely sure that tokens can be obtained. If any issue arises (for example, if Epic's firewall sees requests from a new IP and blocks them), address it. You might need to have the new AWS IP ranges whitelisted by the hospital's IT if they lock down their FHIR API to known clients. Using an ALB means IPs are Amazon's pool, but you could use a NAT gateway for egress with a fixed IP if needed and tell the client that IP.

- **No need to re-approve scopes:** Since the client IDs and secrets aren't changing, the scopes granted remain in place. However, tokens issued before migration might become invalid if domain or sessions change. Anticipate that users might need to re-authenticate to Epic once after migration (if we didn't carry over session state). This is fine – it's not a full re-registration, just a new authorization. You could warn admins that they may need to click "Connect to Epic" again after the migration just to establish a new token, especially if tokens were stored in memory on Replit. But because we persist tokens in the database, we will migrate those too. So if a token was still valid and our app encryption key remains the same, we might even carry on without re-auth (for sandbox at least). In production, rotating keys for security is good, so admins might reconnect anyway as a precaution.

- **Epic App Orchard details:** If the app listing (for a public app) is tied to a specific endpoint, update the listing with new deployment details. For private integration with a hospital's Epic, communicate with their interface team about the upcoming change and verify if any paperwork (like updating the

BAA or integration documents) is needed since the hosting location changed to AWS (usually not, but they might want to know the data is now hosted on AWS under a HIPAA-compliant setup).

- **Post-migration monitoring:** Once on AWS, closely monitor the Epic API usage. The `fhir_rate_limit_per_hour` setting per org is configurable [32] – ensure it's respected and perhaps tune it if needed on AWS side (the app likely already limits calls). Watch for any errors in the token scheduler or API calls that could indicate misconfigurations. If anything fails, be ready to quickly troubleshoot or even temporarily rollback by pointing the domain back to Replit if that instance is still running, to avoid downtime for clients. Having a rollback plan is part of seamless migration (though after thorough testing, rollback might not be needed).

By following these steps, the transition to AWS will be nearly transparent to the end-users and their Epic systems. They shouldn't have to re-register new client apps or re-negotiate integration terms. Our application will appear the same to Epic – just more robust and scalable under the hood.

8. Additional Advanced Improvements and Next Steps

With the app running on AWS and the core multi-tenant and security features in place, we can consider further enhancements to take full advantage of the new environment and meet enterprise-grade requirements:

- **Container Orchestration & Management:** If using ECS, consider using **Infrastructure as Code** (Terraform or CloudFormation) to define the whole stack (VPC, ECS tasks, RDS, etc.). This will help in managing changes, deploying to different regions, or spinning up new isolated stacks for big clients. It also provides documentation of the infrastructure and allows using AWS deployment best practices (blue/green deployments, etc., if needed).
- **Background processing scaling:** Deploy a dedicated **worker container** that runs the RQ worker process. On Replit, the background tasks may have been run in-process or via threads; on AWS we should separate concerns. This worker container will use the same codebase, but instead of running the web server, it runs something like `rq worker fhir_processing` to listen for jobs. We can scale the number of workers for heavy loads (e.g., if one organization triggers a sync of 500 patients, a pool of workers can divide the work). Ensure the task queue names and job timeouts are configured (the app sets 30-45 minute timeouts for some jobs [33] [34], so make sure the worker and Redis parameters allow that).
- **File storage (S3 integration):** Currently, documents (like PDFs from Epic or OCR results) might be stored on the local file system (Replit) or just kept in the database as text. On AWS, use **Amazon S3** for storing any binary files – for example, if the app downloads a PDF from Epic (DocumentReference), instead of storing it on disk, save it to an S3 bucket. The `Document.file_path` field can then contain the S3 object key or URL. S3 is durable and scalable for file storage, and we can serve documents securely from it (with presigned URLs or through our app after verifying permissions). This change will make the app stateless and allow multiple instances to share access to documents. Configure the S3 bucket with encryption and access control (only allow our app's IAM role to read/write, and perhaps set up a lifecycle to expire or transition old files if needed).
- **Caching layer:** For frequently accessed reference data or heavy computations, consider using Redis not just for RQ but also as a cache. For example, caching common queries or the results of recent

FHIR API calls can reduce load and improve response times. Flask-Caching with Redis backend could be added for endpoints that are hit often.

- **Monitoring and alerts:** Enhance monitoring by using AWS CloudWatch Alarms or third-party tools (Datadog, New Relic) to monitor application performance (throughput, error rates). Set up alerts for abnormal situations, e.g., if an organization's Epic sync fails repeatedly (could indicate their credentials expired or API down), send an alert to our support or automatically email the org admin to check their Epic config. We can use the `connection_status` logic in the app to detect issues and notify appropriately.
- **Security audits and testing:** With the AWS environment, schedule regular security tests. This includes vulnerability scanning of the container (check for outdated packages), using AWS Inspector, and annual third-party penetration testing [35]. Also ensure a process for timely patching of any security issues (e.g., if a Python library has a CVE, update the image and redeploy).
- **Compliance documentation:** Now that data is on AWS, update the HIPAA compliance documentation: list AWS as a subprocessor, ensure a BAA with AWS is in place (which it is by using AWS's standard BAA for HIPAA eligible services like EC2, RDS, S3, etc.). Document how data is encrypted, how access is restricted, and have an incident response plan (the TDD mentions a 60-day breach response, which we adhere to).
- **Feature enhancements:** Modularize any organization-specific configurations. For instance, **Preset management** – allow each org to import default screening presets or use the "shared presets" provided by root. This seems partially implemented with org-specific presets vs global [36]. We should test that on AWS with multiple orgs to ensure one org can't see another's presets unless intended as global.
- **Performance improvements:** Aim to meet the performance goals (e.g., <10s to generate a prep sheet) by leveraging the new resources. If needed, increase the instance size or allocate more CPU to the container. Use profiling tools to identify bottlenecks (maybe the OCR or PDF generation is heavy – consider using AWS Lambda for OCR tasks, or an ECS task with GPU if OCR needs acceleration).
- **User experience and support:** Implement administrative tools for support, like the ability for root admin to impersonate an org admin (for support debugging), or to reset an org's Epic connection if needed. Also possibly add an **analytics dashboard** for each org (to show usage stats, etc., which ties to showing ROI).
- **Payment refinement:** Over time, automate more of the sales process – e.g., a public sign-up page where an interested clinic can start a trial by themselves. This could create an Organization record in "trial" status automatically, send an invite to the email they provided, and direct them to enter their Epic sandbox credentials. This self-service model can onboard smaller clients quickly and then they can convert to paid through the app. Keep this in mind as a future capability once the core migration and security improvements are solid.

By following this comprehensive plan, the HealthPrep application will transition from Replit to AWS smoothly and gain enterprise-level robustness. We addressed containerization, AWS setup, database and multi-tenant enhancements, OAuth continuity, security features like 2FA, onboarding flows, and billing integration. These changes set the stage for a scalable, secure SaaS offering that meets both technical and business requirements, without requiring re-registration with Epic for the existing integrations. The system will be positioned for growth, easier maintenance, and higher trust from healthcare clients due to the improved architecture and compliance measures.

Sources:

- HealthPrep V2 system overview and multi-tenancy design [9] [3]

- Previous multi-tenancy implementation notes (Organization model, onboarding flow, RBAC) [37] [23]
- Security and compliance requirements from HealthPrep Technical Design Document [1] [20]
- Epic integration configuration in current system (Epic credentials per org, connection status UI) [27] [30]

---

[1] [2] [20] [31] [35] tdd.txt
https://github.com/mitfusco98/HealthPrepV2/blob/ecd308aa61c685848a68ae5060ad34ca858be31c/attached_assets/tdd.txt

[3] [9] replit.md
https://github.com/mitfusco98/HealthPrepV2/blob/ecd308aa61c685848a68ae5060ad34ca858be31c/replit.md

[4] [11] [36] MULTI_TENANCY_SETUP.md
https://github.com/mitfusco98/HealthPrepV2/blob/ecd308aa61c685848a68ae5060ad34ca858be31c/MULTI_TENANCY_SETUP.md

[5] [6] [12] [13] [14] [16] [17] [23] [24] [37] Pasted--SYSTEM-STRUCTURE-KEY-COMPONENTS-Component-Purpose-Organization-model-Represents-a-tenant-clinic-1755451370621_1755451370622.txt
https://github.com/mitfusco98/HealthPrepV2/blob/ecd308aa61c685848a68ae5060ad34ca858be31c/attached_assets/Pasted--SYSTEM-STRUCTURE-KEY-COMPONENTS-Component-Purpose-Organization-model-Represents-a-tenant-clinic-1755451370621_1755451370622.txt

[7] [8] [15] [18] [19] [21] [22] [25] [26] [32] models.py
https://github.com/mitfusco98/HealthPrepV2/blob/ecd308aa61c685848a68ae5060ad34ca858be31c/models.py

[10] [33] [34] async_processing.py
https://github.com/mitfusco98/HealthPrepV2/blob/ecd308aa61c685848a68ae5060ad34ca858be31c/services/async_processing.py

[27] [28] [29] [30] epic_connection_status.html
https://github.com/mitfusco98/HealthPrepV2/blob/ecd308aa61c685848a68ae5060ad34ca858be31c/templates/admin/epic_connection_status.html